digital
software

OS/8

anguage Reference Manual

Order No. AA-H609A-TA

OS/8

Language Reference Manual

Order No. AA-H609A-TA

ABSTRACT

This document describes the following languages supported by OS/8: BASIC, FORTRAN IV, PAL8, FORTRAN II, FLAP/RALF, SABR.

SUPERSESSION/UPDATE INFORMATION: This manual supersedes sections of the OS/8 Handbook (DEC-S8-OSHBA-A-D).

OPERATING SYSTEM AND VERSION: OS/8 V3D

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, March 1979

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

Copyright © 1979 by Digital Equipment Corporation

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	

CONTENTS

BASIC

FORTRAN IV

PAL8

FORTRAN II

FLAP/RALF

SABR

DOCUMENTATION SET FOR OS/8

OS/8 SYSTEM GENERATION NOTES (AA-H606A-TA)

The System Generation Notes provide the information you need to get a new OS/8 system running.

OS/8 SYSTEM REFERENCE MANUAL (AA-H607A-TA)

The System Reference Manual describes OS/8 system conventions, keyboard commands, and utility programs.

OS/8 TECO REFERENCE MANUAL (AA-H608A-TA)

The TECO Reference Manual describes the OS/8 version of this character-oriented text editing and correcting program.

OS/8 LANGUAGE REFERENCE MANUAL (AA-H609A-TA)

The Language Reference Manual describes all languages supported by OS/8, including BASIC, FORTRAN IV, and the PAL8 assembly language.

OS/8 ERROR MESSAGES (AA-H610A-TA)

This manual lists in alphabetical order all error messages generated by OS/8 system programs and languages.

BASIC

CONTENTS

			Page
CHAPTER	1	OS/8 BASIC	1-1
	1.1	OVERVIEW	1-1
	1.1.1	Writing a BASIC Program	1-1
	1.1.2	The BASIC Character Set	1-2
	1.1.3	Entering and Running a BASIC Program	1-2
	1.2	ELEMENTS OF BASIC	1-3
	1.2.1	Constants	1-3
	1.2.1.1	Numeric Constants	1-3
	1.2.1.2	String Constants	1-4
	1.2.2	Variables	1-5
	1.2.2.1	Numeric Variables	1-5
	1.2.2.2	String Variables	1-5
	1.2.2.3	Subscripted Variables	1-6
	1.2.3	Expressions	1-7
	1.2.3.1	Arithmetic Expressions	1-7
	1.2.3.2	Relational Expressions	1-8
	1.2.3.3	String Concatenation	1-8
	1.3	FORMATTING BASIC STATEMENTS	1-9
	1.4	THE ASSIGNMENT STATEMENT -- LET	1-10
	1.5	THE COMMENT STATEMENT -- REMARK	1-10
	1.6	INPUT AND OUTPUT STATEMENTS	1-11
	1.6.1	INPUT	1-11
	1.6.2	READ, DATA, and RESTORE	1-12
	1.6.3	DIMENSION	1-14
	1.6.4	PRINT	1-16
	1.6.4.1	Printing Zones -- Format Control Characters	1-17
	1.6.4.2	Printing Numbers and Strings	1-17
	1.6.4.3	Printing with the TAB and PNT Functions	1-18
	1.7	CONTROL STATEMENTS	1-19
	1.7.1	Unconditional Transfer -- GOTO	1-19
	1.7.2	Conditional Transfer -- IF GOTO and IF THEN	1-20
	1.7.3	Looping -- FOR, STEP, and NEXT	1-20
	1.7.3.1	Nested Loops	1-22
	1.7.4	Stopping -- END and STOP	1-23
	1.7.5	Jumping to Subroutines -- GOSUB and RETURN	1-23
	1.8	FUNCTIONS	1-24
	1.8.1	Numeric Functions	1-25
	1.8.1.1	Calculating Sine -- SIN	1-25
	1.8.1.2	Calculating Cosine -- COS	1-26
	1.8.1.3	Calculating the Arctangent -- ATN	1-26
	1.8.1.4	Calculating the Tangent	1-27
	1.8.1.5	Finding the Square Root -- SQR	1-27
	1.8.1.6	The Exponential Function -- EXP	1-27
	1.8.1.7	Calculating the Natural Logarithm -- LOG	1-28
	1.8.1.8	The Integer Function -- INT	1-28

		Page
1.8.1.9	The Absolute Value Function -- ABS	1-29
1.8.1.10	The Sign Function -- SGN	1-29
1.8.1.11	Random Numbers -- RND	1-29
1.8.2	String Functions	1-30
1.8.2.1	Finding the Length of a String -- LEN	1-31
1.8.2.2	Finding a Substring -- POS	1-32
1.8.2.3	Displaying a Substring -- SEG$	1-32
1.8.2.4	Converting a Character to ASCII Code -- ASC	1-33
1.8.2.5	Converting ASCII Code to a Character -- CHR$	1-34
1.8.2.6	Converting Numbers from String to Numeric Format -- VAL	1-35
1.8.2.7	Converting a Number to a String -- STR$	1-36
1.8.3	User-Defined Functions	1-36
1.8.3.1	The FNa Function and the DEF Statement	1-36
1.8.3.2	The UDEF Function Call and the USE Statement	1-37
1.8.4	The Debugging Function -- TRC	1-38
1.8.5	Calling for the Date -- the DAT$ Function	1-39
1.9	FILE STATEMENTS	1-40
1.9.1	File Control	1-40
1.9.1.1	Opening a File -- FILE#	1-40
1.9.1.2	Closing a File -- CLOSE#	1-41
1.9.2	File I/O	1-42
1.9.2.1	Reading Data from a File -- INPUT#	1-42
1.9.2.2	Writing Data on a File -- PRINT#	1-43
1.9.2.3	Resetting a File -- RESTORE#	1-44
1.9.2.4	Checking for End-of-File -- the IF END# Statement	1-45
1.10	SEGMENTING PROGRAMS -- THE CHAIN STATEMENT	1-46
1.11	BASIC COMMANDS	1-47
1.11.1	Entering a New Program -- the NEW Command	1-48
1.11.2	Calling for an Old Program -- the OLD Command	1-48
1.11.3	Running a Program -- the RUN Command	1-49
1.11.4	Displaying a Program -- the LIST Command	1-49
1.11.5	Storing a Program -- the SAVE Command	1-50
1.11.6	Renaming a Program -- the NAME Command	1-51
1.11.7	Erasing the Workspace -- the SCRATCH Command	1-51
1.11.8	Leaving BASIC -- the BYE Command	1-51
1.11.9	Resequencing a Program -- Calling RESEQ	1-52
1.11.10	Key Commands	1-52
1.11.10.1	Correcting Typing and Format Errors -- DELETE, CTRL/U	1-53
1.11.10.2	Eliminating Program Lines -- RETURN	1-53
1.11.10.3	Interrupting Program Execution -- CTRL/C	1-53
1.11.10.4	Controlling Program Listings on the Terminal -- CTRL/S, CTRL/Q, and CTRL/O	1-53
CHAPTER 2	CREATING ASSEMBLY LANGUAGE FUNCTIONS	2-1
2.1	INTRODUCTION	2-1
2.2	THE BASIC RUN-TIME SYSTEM - BRTS	2-2
2.2.1	BRTS Symbol Tables	2-3

			Page
	2.2.1.1	The Scalar Table	2-3
	2.2.1.2	The Array Symbol Table	2-3
	2.2.1.3	The String Symbol Table	2-4
	2.2.1.4	The String Array Table	2-5
	2.2.2	String Storage	2-6
	2.2.3	The String Accumulator	2-7
	2.2.4	String Array Storage	2-7
	2.2.5	The DATA List	2-8
	2.2.6	Array Space	2-8
	2.2.7	Compiler Pseudo-Code	2-9
	2.2.8	File Buffer Space	2-9
	2.2.9	Device Handler Space	2-9
	2.2.10	The BRTS I/O Table	2-10
	2.2.11	The BRTS Floating-Point Package	2-10
	2.2.11.1	The Floating-Point Accumulator	2-11
	2.2.11.2	Floating-Point Routines	2-11
	2.2.12	BRTS Overlay Buffer	2-12
	2.3	CALLING FLOATING-POINT ROUTINES	2-12
	2.4	USING BRTS SUBROUTINES IN ASSEMBLY-LANGUAGE FUNCTIONS	2-15
	2.4.1	ARGPRE	2-15
	2.4.2	XPUTCH	2-15
	2.4.3	XPRINT	2-16
	2.4.4	PSWAP	2-16
	2.4.5	UNSFIX	2-17
	2.4.6	STFIND	2-17
	2.4.7	MPY	2-18
	2.4.8	DLREAD	2-18
	2.4.9	ABSVAL	2-18
	2.5	PASSING ARGUMENTS TO THE USER FUNCTION	2-18
	2.5.1	Using the USE Statement	2-19
	2.6	BRTS INPUT/OUTPUT	2-21
	2.7	INTERFACING AN ASSEMBLY LANGUAGE FUNCTION TO BRTS	2-21
	2.8	SOME GENERAL CONSIDERATIONS	2-24
	2.8.1	Routines Unusable by Assembly Language Functions	2-24
	2.8.2	Using OS/8	2-24
	2.8.3	Using Device Driver and File Buffer Space	2-24
	2.8.4	Using the Interrupt Facility	2-24
	2.8.5	Using Page 0	2-25
CHAPTER	3	OPTIMIZING SYSTEM PERFORMANCE	3-1
	3.1	BYPASSING THE BASIC EDITOR	3-1
	3.2	PLACING BASIC OVERLAYS ON THE SYSTEM DEVICE	3-2
	3.3	GROUPING FUNCTION CALLS IN BASIC PROGRAMS	3-2
	3.4	MAKING SAVE IMAGES OF BASIC SOURCE PROGRAMS	3-3
CHAPTER	4	OS/8 BASIC SYSTEM BUILD INSTRUCTIONS	4-1
	4.1	THE BASIC SYSTEM	4-1
	4.2	MAKING SAVE IMAGES FROM BINARY FILES	4-1
	4.2.1	Non-EAE BASIC	4-1
	4.2.2	EAE BASIC	4-2
	4.3	ASSEMBLING THE BASIC SOURCES	4-3

CONTENTS (Cont.)

			Page
CHAPTER	5	LAB8/E FUNCTIONS FOR OS/8 BASIC	5-1
	5.1	GENERAL DESCRIPTION	5-1
	5.2	PREPARING BASIC FOR LAB8/E FUNCTIONS	5-2
	5.3	DEFINITION OF LAB8/E SUPPORT FUNCTIONS	5-2
	5.4	LAB8/E EXAMPLES	5-10
	5.5	GETTING ON THE AIR WITH BASIC	5-19
	5.6	LAB8/E FUNCTION SUMMARY	5-19
APPENDIX	A	SUMMARY OF BASIC EDITOR COMMANDS	A-1
APPENDIX	B	SUMMARY OF BASIC STATEMENTS	B-1
APPENDIX	C	SUMMARY OF BASIC FUNCTIONS	C-1
APPENDIX	D	BASIC ERROR MESSAGES	D-1
INDEX			Index-1

FIGURES

FIGURE	2-1	BRTS Configuration	2-2

TABLES

TABLE	1-1	Alphanumeric Characters and Corresponding ASCII Code Numbers	1-34
	5-1	LAB8/E Function Summary	5-19

CHAPTER 1

OS/8 BASIC

1.1 OVERVIEW

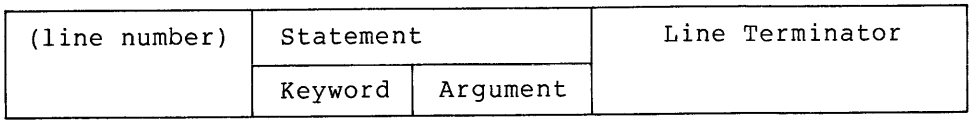
BASIC (Beginner's All-Purpose Symbolic Instruction Code) is a high-level computer language for scientific, commercial, and educational applications.

- BASIC is all-purpose. You can use it to process large amounts of data as well as to solve complex mathematical problems.
- BASIC is conversational. You write programs with simple English keywords and common mathematical expressions. You run, store, and retrieve programs with a set of simple commands resembling English verbs.
- BASIC is interactive. You can input data while a program is running and make changes and corrections in statements under the direction of the BASIC editor. BASIC locates any formatting errors you make in entering your program and prints appropriate messages to help you correct them.

1.1.1 Writing a BASIC Program

You write a BASIC program as a series of numbered lines, each containing one or more instructions called statements.

The format of a typical one-line statement line is



For example:

```
10              PRINT     "HOORAY!"     RETURN
```

The line number -- which may range from 1 to 99999 -- identifies the line and indicates its position in the sequence of operations set out in the program. You do not have to enter the lines in numerical order. BASIC automatically sorts them before it executes the program. You may remove or insert lines at any time to modify a program. For this reason, it is good programming practice to leave room for later additions by numbering lines in increments of five or ten.

The first element in the statement -- the keyword -- tells BASIC what to do. For example, the keyword in the example -- PRINT -- instructs BASIC to display output, such as a message or the result of a computation, on the terminal.

The second element in the statement -- the argument -- may be a formula, a word or phrase, a variable, a line number -- anything BASIC can take action upon. In the example above, the argument is the message "HOORAY!", which BASIC will display on the terminal.

The line terminator -- RETURN -- enters the program line into the system. Even though you type the line on the keyboard and see it echoed on your terminal screen, the BASIC editor does not receive it until you strike the RETURN key.

The last line in every BASIC program must be an END statement. The format is

(line number) END

BASIC statements are described in Sections 1.3 through 1.7.5.

1.1.2 The BASIC Character Set

The alphabet of the BASIC language is the full set of ASCII (American Standard Code for Information Interchange) characters. This set includes

- Upper-case letters A through Z
- Numbers 0 through 9
- Special characters (* and $, for example)
- Nonprinting characters (space and tab, for example)

You may include all ASCII characters in a program. BASIC converts lower-case letters to upper case, ignores nonprinting characters, and leaves all other characters unchanged.

1.1.3 Entering and Running a BASIC Program

To run a BASIC program you must first enter it into a special area in the memory of your computer -- called the workspace -- that BASIC reserves for user-written programs. To do this, summon the BASIC editor by typing

```
.BASIC
```

in response to the Monitor dot. BASIC will then display the message

```
NEW OR OLD --
```

to determine if you want to enter a program from the terminal or run one that you have previously stored as a file.

Assume, for example, that you have a new program called CHEERS that you want to enter and run. Type

```
NEW OR OLD -- NEW CHEERS
```

As soon as BASIC displays a message to indicate that it is READY, begin typing your program line by line.

```
10 FOR K=1 TO 3
20 PRINT "HOORAY!"
30 NEXT K
99 END
```

This complete four-line program now resides in the workspace. To run it, type the command

```
RUN
```

BASIC displays a header line, followed by the program output.

```
HOORAY!
HOORAY!
HOORAY!
```

In addition to the RUN command, BASIC provides commands that let you display on the terminal the program that is in the workspace, store it as a file on a peripheral device and retrieve it later for re-use, change its name, renumber it, or erase it from the workspace. For a complete description of BASIC commands, see Section 1.11.

1.2 ELEMENTS OF BASIC

The following sections define the elements of BASIC programming.

1.2.1 Constants

A constant is a quantity with a fixed value. In BASIC, you may enter constants from the terminal or instruct BASIC to read them from a data list or from a file during program execution.

1.2.1.1 **Numeric Constants** - BASIC accepts numbers within the range

```
10^-616<N<10^+616
```

and treats all numbers as decimal numbers. That is, it accepts any number containing a decimal and assumes a decimal point after any integer.

BASIC uses a second format -- called exponential or E-type notation -- to express numbers outside the range +.00001<N<999999. The format for an E-type number is

```
xxxx.xxxx E(+ or -)nnn
```

where E represents "times 10 to the power of." Thus, for the number 23.4E2, read "23.4 times 10 to the power of 2." Expressed another way,

```
23.4E2=23.4*10**2=2340
```

You may input data in either format. Results of computations with an absolute value outside the range +.00001<N<999999 are always output in E-type format.

BASIC prints six significant digits in normal operation as shown in the following examples.

```
You enter:      BASIC outputs:
.01             .01
.0099           .0099
999999          999999
1000000         1.00000E+006
.0000009        9.00000-007
```

BASIC automatically suppresses leading zeros in integer numbers and trailing zeros in decimal fractions. BASIC outputs exponential numbers in the form

```
(blank or -) x.xxxxxE(+ or -)nnn
```

For example:

```
-3.37021E+008 equals -337,021,000
 7.26000E-004 equals 0.000726
```

BASIC stores numbers internally with a precision of 23 bits. Arithmetic operations are accurate to 22 bits. No rounding is done.

BASIC does conversions from ASCII to internal format and vice-versa in extended precision. Conversion to internal format is rounded to 23 bits. On output, BASIC rounds the result to 6 decimal digits.

1.2.1.2 **String Constants** - A string constant is any keyboard character or group of characters -- letters, numbers, spaces, symbols -- that you want to use as data. In BASIC programs, string constants must be enclosed by quotation marks. The quotation marks instruct BASIC to treat characters within them exactly as you type them in at the terminal.

For example, this program of PRINT statements

```
10 PRINT "I AM A STRING"
20 PRINT "@%$^*& $%&"
30 PRINT "$346.98"
40 PRINT """HI THERE!"""
50 PRINT "30 + 20"
60 PRINT 30 + 20
99 END
```

will cause BASIC to display

```
I AM A STRING
@%$^*& $%&
$346.98
"HI THERE!"
30 + 20
 50
```

Note that BASIC does not consider the enclosing quotation marks to be part of the string. As line 40 demonstrates, to display quotation marks, you must place them within a double pair.

Lines 50 and 60 show the difference between string and numeric data. The quotation marks cause BASIC to display the string "30 + 20" exactly as you enter it. In line 60 BASIC performs a computation on the expression 30 + 20 and prints the sum of 50.

1.2.2 Variables

In BASIC programming, a variable is a symbolic name representing a number or a character string. When you assign a numeric or string value to a variable (with a LET statement, for example), the value is said to be "stored" in the variable. This means that BASIC has placed the value in a memory location -- or locations -- associated with the variable name.

For example, the following statement stores the value 37 in the variable N.

```
LET N=37
```

If N already contains a value, the new value replaces it.

Once you have assigned a value to a variable, BASIC will use the value in any expression in which the variable appears. For example:

```
LET B=N*2
```

This statement evaluates the expression N*2 and stores the result in the variable B.

You may instruct BASIC to change the value of a variable any number of times during one execution of a program. BASIC always uses the most recently assigned value when performing calculations.

The following sections describe numeric variables, string variables, and subscripted variables.

1.2.2.1 **Numeric Variables** - A numeric variable name consists of a letter or a letter followed by a digit. For example,

Acceptable Variables	Unacceptable Variables
M	2C (A variable cannot begin with a digit.)
R2	AB (A variable may contain only one letter.)

Unless you specify otherwise, BASIC automatically sets all variables to zero before executing a program. However, if you wish to assign zero, it is good programming practice to do the initializing yourself at the beginning of the program. You can do this with a series of LET statements or by using READ and DATA statements. For example, this statement

```
10 LET A=0\B=0\C=5
```

tells BASIC to assign 0 to A and B and 5 to C. (See Section 1.6.2 for READ and DATA.)

1.2.2.2 **String Variables** - A string variable name consists of a letter -- or a letter and a digit -- followed by a dollar sign. A$ and A2$ are both legitimate string variable names; 2A$ and AA$ are not.

You may assign no more than eight characters to a string variable unless you have first specified the dimensions with a DIM statement. (See Section 1.6.3.)

1.2.2.3 **Subscripted Variables** - A subscripted variable consists of a string or numeric variable name followed by a subscript in parentheses. A subscript may be a number, a numeric variable, an expression, or any two such elements separated by a comma. The following are all legal subscripted variables:

```
A(2)

A(K)

M(3,4)

M(I,J)

G(K-1)

F$(3.4)
```

If the subscript is not a whole number, BASIC uses only the whole number part. Thus, in the example above F$(3.4) is the same as F$(3). BASIC permits a subscript value of zero.

The subscript in a numeric variable serves as a pointer to a location in a list or table. For example, this subscripted numeric variable

```
X(3)
```

indicates the fourth position in the list

```
X(0), X(1), X(2), X(3), X(4), X(5)
```

Note that all subscripted variables in a list or table share the same variable name.

Two subscripts appended to a numeric variable (and separated by commas) indicate a row and column number in a table. This variable

```
A(3,5)
```

points to row three column five.

One subscript appended to a string variable name indicates the length of the string. Two subscripts indicate its position in a list and its length. For example, this variable

```
R$(35)
```

will accept a string constant 35 characters long. This variable

```
R$(5,35)
```

indicates that the sixth position in a list (beginning with 0) holds a string 35 characters long.

You cannot create a table of string variables in a BASIC program.

A program may contain the same variable in both a subscripted and an unsubscripted form. For example, BASIC will recognize A(1) and A in the same program. However, once subscripted, a variable must contain the same number of subscripts throughout the program. If A(1) occurs, for example, BASIC will not accept A(3,4).

For further discussion of lists and tables, see the DIM statement in Section 1.6.3.

1.2.3 Expressions

An expression is a group of numerics or alphanumerics which, when evaluated, equals a number or a string. Expressions contain special symbols -- called operators -- which direct BASIC in its evaluation. BASIC recognizes three types of operators:

- Arithmetic operators
- Relational operators
- String operators

1.2.3.1 **Arithmetic Expressions** - BASIC uses the following operators to perform addition, subtraction, multiplication, division, and exponentiation.

Symbol	Meaning	Example
+	Addition	A+B
-	Subtraction	A-B
/	Division	A/B
^ or **	Exponentiation	A/B or A**B

In any mathematical formula, BASIC first treats expressions enclosed by parentheses. After parentheses, BASIC maintains the following order of priority.

1. Exponentiation
2. Multiplication and Division (equal priority)
3. Addition and Subtraction (equal priority)

When all of the operators in an expression have equal claim to priority, BASIC simply evaluates the expression from left to right. For example, in this expression,

```
A+B-C
```

BASIC adds A to B and then subtracts C from the sum.

Parentheses let you control the order in which BASIC performs the operations called for in an expression. You may nest parentheses within parentheses. Where nesting occurs, BASIC will give first attention to the elements contained in the innermost "nest."

In this example,

```
A=7*((B**2+4)/X)
```

the order of priority is

1.	B**2	BASIC raises B to the power of 2.
2.	B**2+4	BASIC adds 4 to B**2.
3.	(B**2+4)/X	BASIC divides the result so far by X.
4.	7((B**2+4)/X)	BASIC multiplies by 7 and then assigns the result to A.

Since BASIC ignores spaces, you may use them to make complex expressions easier to read. Spacing will considerably improve the appearance of the example above.

```
A = 7*( (B**2 + 4)/X )
```

1.2.3.2 **Relational Expressions** - Relational operators instruct BASIC to determine the relationship between two values in an expression. BASIC recognizes six relational operators.

=	equal
<	less than
=< or <=	less than or equal to
>	greater than
=> or >=	greater than or equal to
<> or ><	not equal

Relational operators set the conditions in IF-THEN statements. This statement

```
10 IF A>B THEN 50
```

directs BASIC to determine the relationship between A and B and jump to line 50 if A is greater.

You can use strings and string variables in relational expressions. BASIC compares strings one alphanumeric character at a time, using ASCII code numbers to determine if one character is "greater" or "less" than another. BASIC proceeds from left to right until it reaches the end of the strings or until it discovers an inequality. If one string is shorter, BASIC adds spaces to it until both are the same length. For example, in comparing AB to ABCD, BASIC will treat AB as AB (space) (space).

1.2.3.3 **String Concatenation** - BASIC recognizes the ampersand (&) as an operator in string expressions. The ampersand allows you to concatenate strings -- that is, to join them together. For example:

```
10 LET A$="BEAN"
20 LET B$= "TOWN"
30 PRINT A$ & B$
99 END
```

This program will cause BASIC to display:

```
BEANTOWN
```

You may use the ampersand to concatenate strings wherever a string is legal -- with one exception. A concatenated string variable may not appear to the left of the equal sign in a LET statement. Thus, this statement is legal:

```
10 LET A$=B$ & C$
```

this statement is not:

```
10 LET A$ & B$ = C$
```

1.3 FORMATTING BASIC STATEMENTS

Every BASIC program consists of a sequence of numbered lines, each containing one or more instructions called statements.

The format of a typical single-statement line is

```
(line number) keyword argument
```

where the keyword is an instruction to BASIC and the argument is some element that BASIC can act upon.

Here are some examples of single-statement lines.

```
10 PRINT "HOORAY"

20 LET A = 8

45 GOTO 90

80 INPUT R$
```

A multistatement line is one that contains more than one keyword/argument combination. The format is

```
(line number)   STATEMENT1\STATEMENT2\STATEMENT3
```

For example:

```
30   LET X=X + 1 \ PRINT X \ IF X=25 GOTO 80
```

BASIC executes the statements in a multistatement line from left to right. The backslash -- like RETURN -- terminates a statement.

The line number -- which may range from 1 to 99999 -- identifies the line and any statement or statements it contains; it also indicates a line's position in the sequence of operations set out in the program. Keep in mind the following features and rules when entering and numbering BASIC lines.

- You may enter lines in any order. The RUN command causes BASIC to sort all lines into numerical order before executing the program.
- You may add, delete, or shift lines at any time to modify your program.
- You should number lines in increments of five or ten, in order to leave room for additional statements you may want to insert later.
- If your modified program contains consecutively numbered lines, making it difficult to insert further statements, you may renumber your program with the BASIC RESEQ program. The RESEQ program lets you specify a suitable increment between lines.

The keyword -- the first element in the statement -- tells BASIC what it must do in order to successfully execute the instruction. The argument of the statement is the entity that BASIC acts upon. It may be a number, a string, an expression, a variable, or a line number. For example, in the single-statement lines above, the keyword PRINT tells BASIC to display the string HOORAY! on the screen, the keyword LET to assign the value 8 to the variable A, the keyword GOTO to jump to line 90, and the keyword INPUT to receive a value from the terminal and assign it to the variable R$.

1.4 THE ASSIGNMENT STATEMENT -- LET

The LET statement uses the equal sign (=) to assign a value to a variable.

The format is

(line number) [LET] v = expression

where

v	is a variable
expression	is a number, a string, a variable, or an arithmetic expression

The LET statement is the only BASIC statement in which the keyword is optional. For example, these two lines

```
10 LET A = 5
```

```
10 A = 5
```

will both cause BASIC to assign the value 5 to the variable A.

The equal sign in a LET statement indicates replacement rather than equality. That is, the LET statement causes BASIC to evaluate the expression on the right of the equal sign and assign the value to the variable on the left, replacing its previous value. For example, the statement

```
15 K = K + 1
```

causes BASIC to add one to the value of K and store the result in the variable K.

BASIC performs any mathematical operations and functions that you call for in a LET statement. In this statement

```
20 LET A = C + SQR(B)
```

BASIC sets the variable A equal to the value of C plus the square root of the variable B.

This statement

```
5 A$ = "YAZ"
```

assigns a string to a string variable.

The following statement causes BASIC to set element 3,2 in array A equal to element 1,4 in array B.

```
20 LET A(3,2) = B(1,4)
```

1.5 THE COMMENT STATEMENT -- REMARK

The REM statement lets you document your source program with notes and comments, for example:

```
5 REM SUBROUTINE SWAPS VALUES A AND B
```

1.6 INPUT AND OUTPUT STATEMENTS

BASIC provides you with three ways to supply a program with data:

- The INPUT statement lets you type in data while the program is running.
- The READ, DATA, and RESTORE statements let you insert data into the program before you run it.
- BASIC file statements make it possible for you to store data outside the main program and retrieve it under program control.

This section describes only the first two methods. See Section 1.9 for information on file input and output.

The BASIC PRINT statement causes BASIC to display strings and the results of computations on the terminal.

1.6.1 INPUT

The INPUT statement allows you to enter data while the program is running.

The format of the INPUT statement is

```
(line number) INPUT x1, x2,...,xn
```

where x1 through xn represent numeric variables or string variables. If the INPUT statement contains both numeric and string variables, you must enter the appropriate type of data in the proper sequence, assigning numbers to numeric variables and data strings to string variables.

For example, the following line

```
10 INPUT A, B$, C
```

requires a number, a string, and another number entered in that order.

The INPUT statement causes BASIC to pause during the execution of the program, print a question mark (?), and wait for you to type in one value for each variable in the statement. Enter the values, separating them with commas, and press the RETURN key. If you press RETURN without typing in all the data requested, BASIC will display another question mark and await the rest of the data. If you provide more data than the statement requests, BASIC saves the remaining or unused data for use by the next INPUT statement.

BASIC recognizes only the following characters as numeric data.

digits 0 through 9

+ or -

the letter E (for use in floating-point numbers)

. (first decimal point)

BASIC ignores leading spaces and treats all other characters as delimiters for separating numeric data. When BASIC encounters a character other than those specified above, it will assume that it has come to the end of the entry relating to the variable it is currently reading and will apply any character typed in after that to the next variable. Two delimiters in succession signify that the data between delimiters is 0.

For example, the following program requires five numbers:

```
10 INPUT A,B,C,D,E
.
.
.
99 END
```

BASIC prints a question mark to request data.

```
?
```

If, in response to the INPUT prompt, you type

```
-2, 3.7A4E3 9>+1
```

BASIC will assign values to variables in the following manner:

A:-2, B:3.7, C:4000 (4E3=4x10^3=4000), D:9, E:1

BASIC recognizes all characters -- including quotation marks -- as string data and assumes a string length of 8 characters unless you have defined the string variable with a DIM statement. (See Section 1.6.3.) Since it accepts all characters as string data, BASIC treats only the carriage return as the delimiter of a string. To terminate a data string, type the RETURN key.

1.6.2 READ, DATA, and RESTORE

The READ and DATA statements make it possible for you to include data to a program before you run it. During execution, BASIC assigns values listed in the DATA statement to the variables in the READ statement. READ and DATA statements occur only in combination with each other. RESTORE causes BASIC to reuse the values in a DATA statement.

The format of the READ statement is

(line number) READ x1, x2,..., xn

where x1 through xn represent variable names separated by commas.

The format of the DATA statement is

(line number) DATA x1, x2,...,xn

where x1 through xn represent values separated by commas.

Like the INPUT statement, the READ statement must occur in the program before the point where the data is required. DATA statements normally appear at the bottom of the program before the END statement, where you can find them easily when you wish to change input data.

BASIC handles the items in READ and DATA statements sequentially. That is, it assigns the first value in the DATA statement to the first variable in the READ statement, the second variable to the second value, and so on.

A READ statement may contain more or fewer variables than there are values in one DATA statement. READ causes BASIC to search all available DATA statements in the order of their line numbers until it has found values for all variables. When it has assigned values to all of the variables in one READ statement, BASIC will hold the remaining values in the DATA statement until it comes to the next READ statement.

All three of these routines will instruct BASIC to set variable A equal to 1, variable B equal to 2, and variable C equal to 3.

```
10 READ A,B,C
.
.
.
75 DATA 1,2,3
99 END
```

```
10 READ A,B,C
.
.
.
75 DATA 1,2
80 DATA 3
99 END
```

```
10 READ A
.
30 READ B,C
.
.
75 DATA 1,2,3
99 END
```

A DATA statement may contain both string and numeric data. String data in a DATA list must always be enclosed by quotation marks.

This program will cause BASIC to assign 5 to variable C, "AAA" to variable D$, 12 to variable E, and "BEER" to variable F$.

```
10 READ C, D$, E, F$
.
.
.
75 DATA 5, "AAA", 12, "BEER"
```

The RESTORE statement makes it possible for you to use the same data more than once in a program. RESTORE instructs BASIC to reset the data pointer to the first value in the first DATA statement in the program. Since BASIC then proceeds to read through the values as though for the first time, you may use the same variable names on the second pass through the data.

The following program reads a DATA list twice.

```
10 READ A,B,C,D
20 PRINT A;B;C;D
30 RESTORE
40 READ E;F;G;H
50 PRINT E,F,G,H
75 DATA 1,2,3,4
99 END
```

BASIC displays

```
 1  2  3  4
 1  2  3  4
```

1.6.3 DIMENSION

The DIM statement lets you create a list or table of subscripted variables for storing data. (You can organize numeric data in both lists and tables, but BASIC stores strings in lists only.) DIM also defines the length of a string assigned to a string variable.

To create a list -- a one-dimensional array -- of subscripted numeric variables, use the following format:

(line number) DIM x(n)

where

- x — is a numeric variable name. All subscripted variables in the list share the same variable name.
- n — specifies the number of numeric elements in the list. (Since BASIC assigns 0 as the subscript of the first variable, the number of elements in the list is n + 1.)

For a table -- a two-dimensional array -- of subscripted numeric variables, use the form

(line number) DIM x(n,m)

where

- x — is a numeric variable name
- n — specifies the number of rows in the table. (The actual number of rows in the table is n + 1.)
- m — specifies the number of columns. (m + 1 equals the number of columns.)

For example, this DIM statement introduces a list of six subscripted numeric variables:

```
10 DIM A(5)
```

10 DIM A(5)

A(0)	A(1)	A(2)	A(3)	A(4)	A(5)

The following statement describes a table of 24 numeric elements.

```
10 DIM A(3,5)
```

SIX COLUMNS

FOUR ROWS

A(0,0)	A(0,1)	A(0,2)	A(0,3)	A(0,4)	A(0,5)
A(1,0)	A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)
A(2,0)	A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)
A(3,0)	A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)

The number of elements in a table is (n + 1)*(m + 1).

To specify the length of a string, use the DIM statement in the following manner.

```
(line number) DIM X$(n)
```

where

X$ is a string variable

n is the length of the string. A string may contain no more than 72 characters. All strings that exceed 8 characters in length must be dimensioned with a DIM statement.

To introduce a list of subscripted string variables, use the format

```
(line number) DIM X$(n,m)
```

where

X$ is a string variable name

n specifies the number of strings in the list. (The number of strings is n + 1.)

m is the length of each string -- up to 72 characters

For example, this DIM statement describes one string 12 characters long:

```
10 DIM C$(12)
```

This statement describes 4 strings, each 20 characters long:

```
10 DIM D$(3,20)
```

This program will fill variables from a DATA list:

```
10 DIM D$(3,20)
20 FOR Y=0 TO 3
30 READ D$(Y)
40 NEXT Y
50 FOR Z=0 TO 3
60 PRINT D$(Z)
70 NEXT Z
80 DATA "ZERO","ONE","TWO","THREE"
```

Keep in mind the following features and rules concerning the DIM statement:

- Arrays are limited in size only by the amount of available memory -- that is, space not used by the monitor or the program statements.
- Subscripts n and m must be integer numbers. They may not be variables.
- A variable may not appear in a program with subscripts higher than the ones you have described in the DIM statement.
- BASIC assumes a string length of 8 characters or less unless you define the string variable with a DIM statement. If you wish to assign a string that is more than 8 characters long, you must DIMension the string variable.
- BASIC will not accept two-dimensional string variables.
- BASIC assigns a subscript of 0 to the first element in every array. Therefore, the number of elements in a one-dimensional array is n + 1, and the number of elements in a two-dimensional array is (n + 1)*(m + 1).
- You may define more than one array with a single DIM statement. For example, this statement dimensions both the one-dimensional array A and the two-dimensional array B.

```
10 DIM A(20), B(4,7)
```

1.6.4 PRINT

The PRINT statement lets you instruct BASIC to display the results of computations, comments, and the values of variables, or to plot the points of a graph on a terminal.

The format of the PRINT statement is

```
(line number) PRINT expression(s)
```

where expressions are numbers, variables, strings, or arithmetic expressions separated by format control characters. Using the PRINT statement without expressions will output a blank line on the terminal.

To output the result of a computation or the value of a variable at any point in the program, type the line number, PRINT, and the variable name or names separated by a format control character. BASIC will use the current value of the variables to evaluate any algebraic expression in a PRINT statement. Thus, the program

```
10 A=16 \B=5 \C=4
20 PRINT A; (C+B)/3; SQR (A)
99 END
```

will output the following values on the terminal:

```
16  3  4
```

To print a message or comment on the screen, type the text, enclosed by quotation marks, as the expression of a PRINT statement. Use PRINT message statements in combination with INPUT statements to specify the data to be entered.

```
10 PRINT "NUMBER OF SHEEP"
20 INPUT S
```

These lines in a program will produce the following output on the screen.

```
NUMBER OF SHEEP
?
```

PRINT statements may contain a combination of messages and numeric variables. This line

```
50 PRINT "TOTAL NUMBER OF SHEEP ="; T
```

will (assuming that T=354) cause the following to be output during execution of the program:

```
TOTAL NUMBER OF SHEEP = 354
```

1.6.4.1 **Printing Zones -- Format Control Characters** - OS/8 BASIC divides a terminal line into five fixed zones (called print zones) of fourteen columns each. To output data in a five-zone format, separate the variables in the PRINT statement with commas. To output data in a single-space row, separate the variables with semicolons.

The following program illustrates the use of control characters in PRINT statements:

```
10 READ A,B,C
15 PRINT A,B,C,A**2,B**2,C**2
20 PRINT
30 PRINT A;B;C;A**2;B**2;C**2
75 DATA 4;5;6
99 END
RUNNH
 4             5             6             16            25
 36

 4  5  6  16  25  36

READY
```

As this example illustrates, when you list more than five variables in a PRINT statement, BASIC automatically moves the sixth number to the beginning of the next line.

1.6.4.2 **Printing Numbers and Strings** - BASIC prints all numbers (integer, decimal, and E-type) in the following format:

```
sign number space
```

where the sign is either minus (-) or blank and the number is always followed by a blank space.

BASICBASIC prints strings exactly as you type them with no leading or trailing spaces. (To print quotation marks, you must delimit them with a double pair.)

For example:

```
10 PRINT """PRINTING QUOTATION MARKS"""
20 END
RUNNH

"PRINTING QUOTATION MARKS"
```

1.6.4.3 **Printing with the TAB and PNT Functions** - The TAB function allows you to position characters anywhere on the terminal line. You may use the TAB function only in combination with a PRINT statement.

The format of the TAB function is

```
TAB(X)
```

where X is the position (from 1 to 72 columns available on the terminal) in which the next character will be displayed.

Each time the TAB function appears in a PRINT statement, BASIC counts the positions from the beginning of the line, not from the current position of the printing head. For example, the TAB function in the following program causes BASIC to print the character "/" at 24 equally spaced positions across the line.

```
10 FOR K=3 TO 72 STEP 3
20 PRINT TAB(K);"/";
30 NEXT K
99 END
```

If the argument X in the TAB function is less than the current position of the printing head, BASIC starts printing at the current position. If the argument is greater than 72 (the number of columns available in an output line), BASIC executes a carriage return and a line feed and then resumes printing at position 1.

The PNT function allows you to perform special nonprinting actions on the terminal, such as ringing the buzzer, erasing the screen, moving the cursor, etc.

The format of the PNT function is

```
PNT(X)
```

where the argument X represents the decimal value of the 7-bit ASCII character to be output.

For example, to ring the buzzer on the terminal, type

```
10 PRINT PNT(07)
```

1.7 CONTROL STATEMENTS

During the execution of a program, BASIC ordinarily passes from one line to the next in ascending numerical order. BASIC control statements make it possible for you to alter the normal sequence -- either unconditionally or only when certain conditions are met. Thus, you can:

- repeat a set of statements
- skip statements
- stop and check values
- terminate a program

This section describes the statements that allow you to change the normal sequence of statement execution.

1.7.1 Unconditional Transfer -- GOTO

The GOTO (or GO TO) statement causes BASIC to jump to any line in the program that you specify. The GOTO statement sets no conditions.

The format of the GOTO statement is

(line number) GOTO n

where n is the number of the line to which BASIC will jump.

When BASIC encounters a GOTO statement, it jumps immediately to the line beginning with the number indicated. For example, this program

```
10 GOTO 40
20 PRINT "SECOND"
30 STOP
40 PRINT "FIRST"
50 GOTO 20
99 END
```

will display

```
FIRST
SECOND
```

If you specify a nonexecutable statement (such as REM) in a GOTO line, BASIC will proceed to the next executable statement.

NOTE

If you inadvertently create an infinite loop with a GOTO statement, halt BASIC with the CTRL/C command.

1.7.2 Conditional Transfer -- IF GOTO and IF THEN

IF GOTO and IF THEN statements use relational operators to test for a specified relationship between two variables, numbers, strings, or expressions. When the relational expression is true, BASIC executes the GOTO instruction. When the IF statement is false, BASIC proceeds to the next line in the program.

The format of the IF GOTO (or IF THEN) statement is

(line number) IF v1 relation v2 GOTO x

where

v1 and v2	represent variable names, numbers, strings, or expressions
relation	is any relational operator
x	is the number of the line to which BASIC will jump if the relation is true

This example

```
10 LET A=5
20 IF A=2 GOTO 99
30 PRINT "NO"
99 END
```

will cause BASIC to display

NO

BASIC compares strings one alphanumeric character at a time, using ASCII code numbers to determine if one character is "greater" or "less" than another. BASIC proceeds from left to right until it reaches the ends of the strings or until it finds an inequality. If one string is longer than the other, BASIC adds spaces to the shorter string until both are the same length. For example, in comparing AB to a four-letter string, BASIC will treat AB as "AB (space) (space)".

1.7.3 Looping -- FOR, STEP, and NEXT

Programs frequently require the repetition of some instruction or sequence of instructions. One way to achieve this is to write out the steps as many times as you wish BASIC to execute them. For example, this program

```
10 PRINT "HOORAY!"
20 PRINT "HOORAY!"
30 PRINT "HOORAY!"
99 END
```

will instruct BASIC to display HOORAY! in three lines on the terminal.

A better way to achieve the same end is to write the PRINT statement once and instruct BASIC to run through it three times. This type of repetition, which requires BASIC to jump backward in the program and retrace its steps, is called looping.

To execute a loop in a program, BASIC must know two things: which statements to repeat, and how many times to repeat them. FOR and STEP statements let you supply this information.

The format is

```
(line number) FOR v=x TO y[STEP z]
```

where

v	is a variable name. It is the index of the loop, increased or decreased each time the loop is executed.
x	is an expression (numerical value, variable name, or mathematical expression) indicating the initial value of the index -- that is, the value of v before the loop is executed the first time.
y	is an expression indicating the terminal value of the index -- the value of v after the last execution of the loop.
STEP z	is an optional statement used to specify the increment. If you omit it, BASIC assumes a STEP value of 1.

For example, this statement

```
15 FOR K=2 TO 20 STEP 2
```

tells BASIC to repeat the loop as long as K is less than or equal to 20. Since K is incremented by 2 after each execution, BASIC will run through the loop 10 times.

The NEXT statement marks the end of a program loop. It occurs only in combination with a FOR statement.

The format of the NEXT statement is

```
(line number) NEXT v
```

where v is the index variable in the FOR statement.

The NEXT statement causes BASIC to add the STEP value to the index (or to add 1 if the FOR statement contains no STEP value) and to check to see if the value of the index exceeds the terminal value. If it does, BASIC falls through the loop and executes the line following the NEXT statement.

To cause BASIC to exit from a loop before the index has reached the terminal value, use an IF-THEN statement. BASIC can reenter only those loops that it has left before completion.

NOTE

Do not attempt to transfer control from a loop to a subroutine located above it in the program. Doing so may cause BASIC to execute the loop a wrong number of times.

The following example shows one way to use a FOR-NEXT loop to produce the same results of the HOORAY! program above.

```
10 FOR K=2 TO 6 STEP 2
15 PRINT "HOORAY!"
20 NEXT K
99 END
```

The FOR statement tells BASIC to repeat the loop as long as K is less than or equal to 6. Since K is incremented by 2 after each execution, BASIC will run through the loop three times.

1.7.3.1 **Nested Loops** - You may place one or more loops within a loop provided that the inner loops are completely contained by the outer and that no overlapping of loops occurs. Placing one loop within another is called nesting. Each nested loop must have its own FOR and NEXT statements and must terminate before the loop that contains it.

The following examples show legal and illegal types of nested loops:

Legal

```
10 FOR A=1 TO 10
20 FOR B=2 TO 20
30 NEXT B
40 NEXT A
```

Legal

```
10 FOR A=1 TO 10
20 FOR B=2 TO 20
30 NEXT B
40 FOR C=3 TO 30
50 FOR D=4 TO 40
60 FOR E=5 TO 50
70 NEXT E
80 NEXT D
90 NEXT C
95 NEXT A
```

Illegal

```
10 FOR M=1 TO 10
20 FOR N=2 TO 20
30 NEXT M
40 NEXT M
```

The following program contains a nested loop:

```
10 PRINT "INNER",,"OUTER"
15 PRINT
20 FOR I=1 TO 2
30 FOR O=1 TO 3
40 PRINT I,,O
50 NEXT O
60 NEXT I
99 END
```

BASIC will execute the loops and display:

INNER	OUTER
1	1
1	2
1	3
2	1
2	2
2	3

Note that each execution of the outer loop causes BASIC to run through the inner loop three times.

1.7.4 Stopping -- END and STOP

Two statements -- END and STOP -- will cause BASIC to terminate the execution of a program and return control to the editor.

The END statement informs the BASIC compiler that it has come to the last line in the program. Every BASIC program must end with an END statement. No program may contain more than one END statement. A STOP statement cannot take the place of the END statement.

The format of the END statement is

(line number) END

which causes BASIC to return to the edit mode, display

READY

and await your next command.

The STOP statement also terminates a running program, but unlike END, it may occur more than once in the same program.

The format of the STOP statement is

(line number) STOP

The following program demonstrates the use of the STOP statement:

```
10 INPUT A
20 READ B
30 IF A=B GOTO 50
40 STOP
50 PRINT "EQUAL"
60 DATA 3
99 END
```

The STOP statement here prevents BASIC from displaying EQUAL when A does not equal B.

1.7.5 Jumping to Subroutines -- GOSUB and RETURN

A subroutine is a sequence of statements that performs some operation required at more than one point in the program. Subroutines are generally placed at the end of the program, usually before any DATA lines and always before the END statement.

Two statements -- GOSUB and RETURN -- cause BASIC to jump to a subroutine, execute it, and jump back to the point in the main program where it left off. GOSUB and RETURN occur only in combination with each other.

The format of the GOSUB statement is

(line number) GOSUB n

where n is the number of the first line in the subroutine.

When BASIC encounters a GOSUB, it records the number of the line immediately following it and jumps to the first line of the subroutine.

The format of the RETURN statement is

(line number) RETURN

The RETURN statement always occupies the last line in the subroutine. RETURN causes BASIC to jump to the line following the last GOSUB statement it has executed.

You may use the control statements described in this chapter to direct BASIC from one line to another within a subroutine or even to a line in another subroutine.

You may also "nest" subroutines -- use one subroutine to call another -- up to ten levels. If you exceed the tenth level, BASIC prints

GS AT LINE y

where y represents the line number where the error occurred.

The following sample program contains two simple subroutines:

```
10 GOSUB 60
20 PRINT "I'M BACK FROM 1"
30 GOSUB 80
40 PRINT "I'M BACK FROM 2"
50 STOP
60 PRINT "SUBROUTINE 1"
70 RETURN
80 PRINT "SUBROUTINE 2"
90 RETURN
99 END
```

The STOP statement prevents BASIC from "falling into" the subroutines and executing them after it has executed the PRINT statement in line 40. The program will produce:

```
SUBROUTINE 1
I'M BACK FROM 1
SUBROUTINE 2
I'M BACK FROM 2
```

1.8 FUNCTIONS

Functions are special subroutines that perform frequently used operations on numbers and strings.

The format of most functions is

NNN(X)

where

NNN	is a three-letter name
(X)	is an argument enclosed in parentheses. The argument may be a number, a variable, an expression, or another function.

Some functions require multiple arguments and take the form

NNN(X,Y,Z)

Most functions compute a value based on the value of the argument or arguments involved. They are said to "return" this value. For example, SQR(Z) returns the square root of Z.

Functions may return either strings or numbers. Functions that return character strings are distinguished from functions that return numbers by the dollar sign ($) appended to their name. For example, the CHR$ function converts an ASCII code number to its equivalent character and returns the character. The ASC function converts a character to its code number.

Unlike conventional subroutines, functions do not require GOSUB and RETURN statements. They produce their results "in place." For example, the following line will assign the variable A a value of 2:

```
10 LET A=SQR(4)
```

1.8.1 Numeric Functions

BASIC provides numeric functions to perform standard mathematical operations. For example, you may find it necessary to find the sine of an angle. You can do this by looking it up in a table of sine values or by using the BASIC SIN function.

BASIC provides the following trigonometric functions:

- Sine function (SIN)
- Cosine function (COS)
- Arctangent function (ATN)

BASIC provides algebraic functions to find:

- the square root of a number (SQR)
- the value of e -- 2.71828 -- raised to any power (EXP)
- the natural logarithm of a number (LOG)
- the integral part of a number (INT)
- the absolute value of a number (ABS)
- a value based on the sign of a number (SGN).

BASIC also includes a function -- RND -- that returns a random number. You can use this function when you are trying to simulate an unpredictable situation with a BASIC program.

1.8.1.1 **Calculating Sine -- SIN** - The BASIC SIN function lets you calculate the sine of an angle specified in radians. The format is

```
SIN(X)
```

where

X	is a number, numeric variable, expression, or another function, representing the size of an angle in radians

For example, this program

```
10 LET P = 3.14159
20 PRINT SIN(30*P/180)
30 END
```

will display:

```
0.5
```

1.8.1.2 **Calculating Cosine -- COS** - The BASIC COS function lets you calculate the cosine of an angle specified in radians. The format is

```
COS(X)
```

where

X	is a number, numeric variable, expression, or another function, representing the size of an angle in radians

Thus, these lines

```
10 PRINT COS(45*3.14159/180)
20 END
```

will display

```
0.707108
```

1.8.1.3 **Calculating the Arctangent -- ATN** - The BASIC ATN function lets you calculate the angle (in radians) whose tangent is given as the argument of the function.

The format is

```
ATN(X)
```

where

X	is a number, variable, expression, or another function representing the tangent of an angle

Thus, this two-line program

```
10 PRINT ATN(.57735)
20 END
```

will display

```
0.523598
```

1.8.1.4 **Calculating the Tangent** - Although BASIC does not provide a tangent function, you can find the tangent of an angle with the following trigonometric equation:

$$\text{tangent (angle)} = \frac{\text{sine (angle)}}{\text{cos (angle)}}$$

Translated into BASIC, this equation will read

```
10 T=SIN(R)/COS(R)
```

where T is the tangent and R is an angle expressed in radians.

1.8.1.5 **Finding the Square Root -- SQR** - The BASIC SQR function computes the positive square root of an expression. The format is

```
SQR(X)
```

where

X	is a number, variable, expression, or another function

If the argument is negative, the absolute value of the number is used. For example, this program

```
10 PRINT SQR(16)
20 PRINT SQR(-4)
30 END
```

will display

```
 4
 2
```

1.8.1.6 **The Exponential Function -- EXP** - The BASIC EXP function calculates the value of e raised to the X power, where e is equal to 2.71828. That is, EXP(X) is equivalent to 2.71828**X.

The format is

```
EXP(X)
```

where

X	is a number, numeric variable, expression, or another function

Thus, this program

```
10 PRINT EXP(1.5)
20 END
```

will display

```
4.48169
```

1.8.1.7 **Calculating the Natural Logarithm -- LOG** - The BASIC LOG function calculates the natural logarithm of X (to the base e).

The format is

```
LOG(X)
```

where

X	is a number, numeric variable, expression, or another function

EXP and LOG perform opposite functions. That is the exponent x (the input in the EXP function) in the formula e^x=y is the logarithm of y to the base e (the output of the LOG function) in the formula x=log(e) y.

This BASIC formula demonstrates their relationship:

```
LOG(EXP(X)) = X
```

1.8.1.8 **The Integer Function -- INT** - The BASIC INT function returns the value of the largest integer not greater than the argument. The format is

```
INT(X)
```

where

X	is a number, numeric variable, expression, or another function

To round off a number to the nearest integer, specify INT(X+.5).

For example, this function

```
10 INT(34.67)
```

returns the value 34;

these functions

```
10 INT (34.67 + .5)
15 INT (34.36 + .5)
```

return the values 35 and 34;

this function

```
10 INT (-14.37)
```

returns the value -15.

1.8.1.9 **The Absolute Value Function -- ABS** - The BASIC ABS function returns the absolute value of an expression. The format is

```
ABS(X)
```

where

X is a number, numeric variable, or numeric expression

By mathematical definition, the absolute value of a number which represents its magnitude is always positive. The absolute value of a positive number is equal to the number; the absolute value of a negative number is equal to the number times -1. For example, this program:

```
10 PRINT ABS(-5)
20 END
```

will display

```
5
```

1.8.1.10 **The Sign Function -- SGN** - The SGN function lets you determine if an expression is positive, negative, or equal to zero. The format is

```
SGN(X)
```

where

X is a number, numeric variable, expression, or another function

If the argument is any positive number, the SGN function will return a value of 1. If the argument is negative, SGN returns -1. If it is 0, SGN returns 0. For example, these lines

```
10 LET A=5\LET B=0\LET C=-2
20 PRINT SGN(A);  SGN(B);  SGN(C)
```

will display

```
1  0  -1
```

because

```
5>0,
0=0, and
-2<0.
```

1.8.1.11 **Random Numbers -- RND** - A random-number series is a series of numbers that are not related to each other in any way. You can use random numbers in a BASIC program to simulate a situation in which the outcome is not predictable -- the flip of a coin, for example, or the rolling of dice.

It is not possible to produce a series of truly random numbers on a computer since, given the same starting conditions, a computer always comes up with the same results. Instead, BASIC uses complex calculations to generate a series of numbers that seem unrelated. This is called a pseudo-random series.

The BASIC RND function produces pseudo-random numbers between -- but not including -- 0 and 1. The format is

```
RND(X)
```

where

X is a dummy variable. Type the function just as it appears above.

Each time BASIC encounters the RND function in a program, it produces a different decimal number. However, if you run the program again, BASIC will output the same set of numbers. To generate a different set of numbers with each execution, use the RANDOMIZE statement in your program.

The format of the RANDOMIZE statement is

```
(line number) RANDOMIZE
```

The following routine will print a different series of random numbers each time you run it. (RANDOMIZE uses the value you enter to vary the output.)

```
10 INPUT X
20 FOR L=1 TO 20
30 PRINT INT (10*RND(X));
40 NEXT L
50 IF X <0 GOTO 20
99 END
```

1.8.2 String Functions

BASIC string functions let you examine and modify strings and perform certain conversions between numbers and strings. Functions that return strings are distinguished from functions that return numbers by the dollar sign ($) after their name.

BASIC provides three functions that allow you to analyze and manipulate strings:

- LEN function -- determines the length of a string
- POS function -- searches for the position of a set of characters within a string
- SEG$ function -- copies a segment from a string

Other functions enable you to convert strings to numbers and numbers to strings:

- ASC function -- converts a character to its ASCII code equivalent
- CHR$ -- converts an ASCII code number to a character
- STR$ -- converts a number to its string representation
- VAL -- converts a string representation of a number to a number

1.8.2.1 **Finding the Length of a String -- LEN** - The LEN function returns the number of characters in a string.

The format is

```
LEN(X$)
```

where

X$ is a string, a string variable, or several concatenated strings and/or string variables

For example,

(1) This line:

```
10 PRINT LEN("DOG")
```

will display

3

(2) This program:

```
10 A$="UP, "
20 B$="DOWN, AND "
30 PRINT LEN(A$&B$&"AROUND")
40 END
```

will display

20

because

```
"UP," =          4 characters
"DOWN, AND" =   10 characters
"AROUND" =       6 characters

Total           20 characters
```

If a string has never been defined, it will have a length of 0.

This program:

```
20 PRINT LEN(L$)
99 END
```

will display

0

1.8.2.2 **Finding a Substring -- POS** - The BASIC POS function returns the location of a specified group of characters in a string.

The format is

```
POS(X$,Y$,Z)
```

where

X$	is the string you want to search
Y$	is the substring you are searching for
Z	is the position in the string at which you want to begin the search

This function searches X$ for the first occurrence of Y$. It begins the search with the Zth character in X$. Depending on what it finds, POS returns the following results.

1. If it finds substring Y$, POS returns the position of the first character in the series.
2. If it fails to find Y$, POS returns a 0.
3. If Y$ is a null string (containing no characters), POS returns a 1.
4. If X$ is a null string, POS returns a 0.

NOTE

If Z is less than 0 or greater than the string, BASIC prints an error message and stops the program.

These lines cause the POS function to start at the seventh character in the string "ABCDEFGHIDEF" and search for the substring "DEF":

```
10 DIM B2$(12)
20 B2$ = "ABCDEFGHIDEF"
30 PRINT POS (B2$, "DEF", 7)
```

POS returns 10. (Change the 7 to a 1 in line 30 and POS will return a 4.)

1.8.2.3 **Displaying a Substring -- SEG$** - The SEG$ function searches for a segment -- a substring -- of a string and returns it for display.

The format is

```
SEG$(X$,Y,Z)
```

where

X$	is the string containing the substring you want to display. X$ may be a variable or the string itself.
Y	is the position of the first character in the substring
Z	is the position of the last character in the substring

SEG$ returns a null string (no characters) if:

- Y is greater than the length of X
- Z is less than 1
- Z is less than Y

If Y is less than 1, SEG$ sets it to 1. If Z is greater than the length of X$, SEG$ sets it equal to the length of X$.

These lines

```
10 DIM B2$(12)
20 B2$ = "ABCDEFGHIDEF"
30 PRINT SEG$(B2$,3,5)
```

will display:

```
CDE
```

1.8.2.4 **Converting a Character to ASCII Code -- ASC** - The ASC function converts a one-character string to its ASCII code equivalent. The format is

```
ASC(X)
```

where

X	is a one-character string

ASC returns the equivalent decimal number for the argument. Table 1-1 lists all the alphanumeric characters available on the terminal and their ASCII code numbers.

Table 1-1
Alphanumeric Characters and Corresponding ASCII Code Numbers

Character	Decimal	Character	Decimal
@	0	(space)	32
A	1	!	33
B	2	"	34
C	3	#	35
D	4	$	36
E	5	%	37
F	6	&	38
G	7	'	39
H	8	(	40
I	9	)	41
J	10	*	42
K	11	+	43
L	12	'	44
M	13	-	45
N	14	.	46
O	15	/	47
P	16	0	48
Q	17	1	49
R	18	2	50
S	19	3	51
T	20	4	52
U	21	5	53
V	22	6	54
W	23	7	55
X	24	8	56
Y	25	9	57
Z	26	:	58
[	27	;	59
\	28	<	60
]	29	=	61
^	30	>	62
_	31	?	63

Thus, this program

```
10 LET A$="*"
20 PRINT ASC("P"),ASC(A$),ASC(")
30 END
```

will display

```
 16          42          57
```

1.8.2.5 **Converting ASCII Code to a Character -- CHR$** - The CHR$ function converts a code number to its equivalent character.

The format is

```
CHR$(X)
```

where

X is a number, a numeric expression, or a numeric variable

CHR$ returns the equivalent character for the argument. (See the ASC function for the table of decimal/character conversions.)

If the argument is greater than 63, divide it by 64 and use the remainder to search the table.

Thus, this line:

```
10 PRINT CHR$(1),CHR$(40)
```

will display

```
A       (
```

This line:

```
10 PRINT CHR$(207),CHR$(77)
```

will display

```
O       M
```

207/64 = 3, with a remainder of 15
 77/64 = 1, with a remainder of 13

Using 15 and 13 to search the table yields the letters "O" and "M".

1.8.2.6 **Converting Numbers from String to Numeric Format -- VAL** - The VAL function converts numbers in string form to numeric data. The format is

```
VAL(X$)
```

where

X$ is a string made up of those values that BASIC accepts as numeric data. These are:

- digits 0 through 9
- + or - sign
- the letter E
- leading spaces -- BASIC ignores them
- the first decimal point (.)

Keep in mind that BASIC does not consider numbers and numeric expressions in string form as numeric data. It will not use them in calculations or as arguments in mathematical functions until you convert them into numeric format with the VAL function.

This program instructs BASIC to read a string, convert it into numeric form, and multiply it by two:

```
10 INPUT A$;
20 PRINT VAL(A$)*2
30 END
```

BASIC displays:

```
?2.46111
 4.92222
```

1.8.2.7 **Converting a Number to a String -- STR$** - The STR$ function converts numerics to strings. The format is

```
STR$(X)
```

where

X	is a numeric expression

The STR$ function returns the string value of the expression exactly as BASIC would print it but without a leading or trailing space. Use the STR$ function when you want to print a number without a leading or trailing space and when you want to perform string operations or functions on a number.

1.8.3 User-Defined Functions

1.8.3.1 **The FNa Function and the DEF Statement** - In some programs, you may want to perform the same sequence of string or numeric operations more than once. As an aid in such cases, BASIC lets you define your sequence as a special function -- called a user-defined function -- that you can call for in the same way you would call for any string or numeric function that BASIC provides.

The BASIC DEF statement lets you create user-defined functions. The format of the DEF statement is

```
(line number) DEF FNa (list) = expression
```

where

(list)	contains the dummy variable or variables that appear in your operation. The same variables must appear in the expression.
expression	is the operation you want BASIC to perform each time you call for the function. The operation may contain numbers, several variables, other functions, or mathematical expressions.

For example, if you write a program in which you repeatedly use the operation e^-x2+5, you can introduce it as a user-defined function with this DEF statement:

```
30 DEF FNE(X)=EXP(-X**2)+5
```

and then call for various values of the function -- FNE(.1), FNE(3.45), FNE(A+2), etc.

This statement:

```
10 DEF FNA(S)=S**2
```

will cause the user-defined function in this line

```
20 LET R=FNA(4)
```

to return a 16.

If the function involves more than one variable, BASIC will identify them by their position. For example, this program

```
10 DEF FNH(N,P)=2*P+N
20 LET X=4\LET Y=5
30 PRINT FNH(X,Y)
40 END
```

will display

```
14
```

BASIC takes the first value in the function (4) as "N", because "N" appears first in the DEF statement. It takes the second value (5) as "P", because "P" is in the second position.

```
DEF FNH (N,P) = 2*P+N

first position          second position


PRINT FNH(X,Y)
```

You must introduce each user-defined function with a separate DEF statement, taking care to place each DEF statement before the first occurrence of the function it defines. For example, if you want to use a special function called FNB(X) in your program, you must first write a DEF statement with FNB as the parameter. You may define up to 26 FN functions in the same program (FNA, FNB...,FNZ).

1.8.3.2 **The UDEF Function Call and the USE Statement** - OS/8 BASIC lets you add one or more user-coded assembly-language functions to a BASIC program and use them in the same way you would use any other function. For complete instructions to write and interface such functions, see Chapter 2.

To specify a user-coded function in an OS/8 BASIC program, type

```
line number UDEF function name(a,b,c)
```

where

function name	consists of alphabetic characters only and has at least one argument (a dummy, if necessary)
(a,b,c)	are arguments. User-written assembly-language functions may contain up to four numeric and two string arguments.

For example:

```
10 LET R=4
15 LET B=6
20 LET Q=10
25 UDEF PLT(X,Y,Z)
30 LET D=PLT(R,B,Q)
35 PRINT 4*D
40 END
```

Line 25 introduces the function PLT to OS/8 BASIC and indicates the number and type of arguments associated with it. In line 30 the function appears as any standard function might appear in a BASIC program. If the function requires an array, a USE statement identifying the array must precede the statement that calls the function. Thus:

```
10 DIM S(15,5)
 .
 .
20 LET Q=10
22 USE S
25 UDEF PLT(X,Y,Z)
 .
 .
 .
```

1.8.4 The Debugging Function -- TRC

The TRC function causes BASIC to print the line numbers of statements in a program in the order that it executes them. This lets you follow the course of loops and subroutines and provides a useful tool for debugging a program.

The format of the TRC function is

```
v = TRC(X)
```

where

v	is any letter. It has no purpose except to occupy the position in the line.
X	is 1 or 0. 1 turns the function on; 0 turns it off.

When it comes upon a TRC(1) in a program, BASIC begins displaying the line number (enclosed by percent signs) of each statement it executes -- with the exception of the following types: DATA, DEF, DIM, END, GOTO, NEXT, RANDOMIZE, REM, and STOP. Encountering a TRC(0) will cause it to stop outputting line numbers and resume normal operation.

For example, this program:

```
60 T=TRC(1)
70 GOSUB 90
80 GOTO 140
90 PRINT "IN OUTER SUB"
100 GOSUB 120
110 RETURN
120 PRINT "IN INNER SUB"
130 RETURN
140 T=TRC(0)
150 END
```

will display

```
% 70 %
% 90 %
IN OUTER SUB
% 100 %
% 120 %
IN INNER SUB
% 130 %
% 110 %
% 140 %
```

1.8.5 Calling for the Date -- the DAT$ Function

The DAT$ function returns the current system date.

The format is

```
DAT$(X)
```

where

X is a dummy variable

Enter this function exactly as it appears above. DAT$ returns an eight-character string in the form

```
mm/dd/yy
```

For example, these lines:

```
10 LET D$ = DAT$(X)
20 PRINT D$
```

will display

```
07/20/77
```

if that date was entered with the monitor DATE command.

If you have not specified the date with the MONITOR date command, the function will return no characters.

1.9 FILE STATEMENTS

BASIC file statements -- which are distinguished from other BASIC statements by the number sign (#) -- let you store data on peripheral devices for later use in any BASIC program. They include:

- FILE# Describes the file, assigns it a channel number from 1 to 4 (the number of files that BASIC can handle at one time), and opens it.
- INPUT# Reads data from the file.
- PRINT# Writes data on the file.
- RESTORE# Resets the pointer to the beginning of the file.
- CLOSE# Closes the file and removes the channel number.
- IF END# Tests for end-of-file.

In most operations, you open a file (FILE#) for input or output, read from it (INPUT#) or write on it (PRINT#), and close it (CLOSE#). You may open only four files at a time -- excluding the terminal, which is always open and available for use. However, the ability to open and close files under program control gives you access to an unlimited number of files. That is, when you close a file, you may reassign its channel number to a newly opened file.

BASIC treats files in the same way it treats terminal input and output. The INPUT statement causes BASIC to read a value that you enter on the terminal and assign it to a variable; the INPUT# statement causes it to read a value from a file. The PRINT statement instructs BASIC to display data on the terminal; the PRINT# statement tells it to write data in a file.

BASIC uses two types of file: string files and numeric files. You may write numbers into a string file in both string and numeric format. Numeric files, however, may contain numeric data only.

1.9.1 File Control

You must open a file with a FILE# statement before you can read or write any data. You should close any files that you open during the course of a program with a CLOSE# statement. The CLOSE# statement cancels the channel number that you have assigned with the FILE# statement, making the channel available to any other newly opened file.

1.9.1.1 **Opening a File -- FILE#** - The BASIC FILE# statement opens a file for input or output, defines it, and assigns a channel number. An input file is one you are reading from. An output file is one you are writing to.